# No One (Cluster) Size Fits All: Automatic Cluster Sizing for Data-intensive Analytics

Herodotos Herodotou
Duke University
hero@cs.duke.edu

Fei Dong
Duke University
dongfei@cs.duke.edu

Shivnath Babu[*]
Duke University
shivnath@cs.duke.edu

## ABSTRACT

Infrastructure-as-a-Service (IaaS) cloud platforms have brought two unprecedented changes to cluster provisioning practices. First, any (nonexpert) user can provision a cluster of any size on the cloud within minutes to run her data-processing jobs. The user can terminate the cluster once her jobs complete, and she needs to pay only for the resources used and duration of use. Second, cloud platforms enable users to bypass the traditional middleman—the system administrator—in the cluster-provisioning process. These changes give tremendous power to the user, but place a major burden on her shoulders. The user is now faced regularly with complex *cluster sizing problems* that involve finding the cluster size, the type of resources to use in the cluster from the large number of choices offered by current IaaS cloud platforms, and the job configurations that best meet the performance needs of her workload.

In this paper, we introduce the *Elastisizer*, a system to which users can express cluster sizing problems as queries in a declarative fashion. The Elastisizer provides reliable answers to these queries using an automated technique that uses a mix of job profiling, estimation using black-box and white-box models, and simulation. We have prototyped the Elastisizer for the Hadoop MapReduce framework, and present a comprehensive evaluation that shows the benefits of the Elastisizer in common scenarios where cluster sizing problems arise.

## Categories and Subject Descriptors

H.3.4 [**Information Storage and Retrieval**]: Systems and Software—*Distributed systems*

## General Terms

Performance

## Keywords

Cluster Provisioning, MapReduce, Cloud Computing

[*]Supported by NSF grants 0644106 and 0964560

## 1. INTRODUCTION

Timely and cost-effective analytics over large datasets is now a crucial requirement in many businesses and scientific disciplines. Web search engines and social networks log every user action. The logs are analyzed at frequent intervals to detect fraudulent activity, to find opportunities for personalized advertising, and to improve the Web-site structure and content. Scientific fields like biology and economics as well as traditional areas like journalism have fast growing computational subareas.

The MapReduce framework—which consists of a programming model and a scalable and fault-tolerant run-time system—is now a popular platform for data analytics [10]. *Hadoop* is an open-source implementation of MapReduce used in production deployments. Hadoop is used for applications like log file analysis, Web indexing, report generation, machine learning research, financial analysis, scientific simulation, and bioinformatics.

Cloud platforms make MapReduce an attractive proposition for small organizations that need to process large datasets, but lack the computing and human resources of a Google, Microsoft, or Yahoo! to throw at the problem. *Elastic MapReduce*, for example, is a hosted service on the Amazon cloud platform where a user can instantly provision a Hadoop cluster running on any number of *Elastic Compute Cloud (EC2)* nodes [1]. The cluster can be used to run data-intensive MapReduce jobs, and then terminated after use. The user has to pay (only) for the nodes provisioned to the cluster for the duration of use.

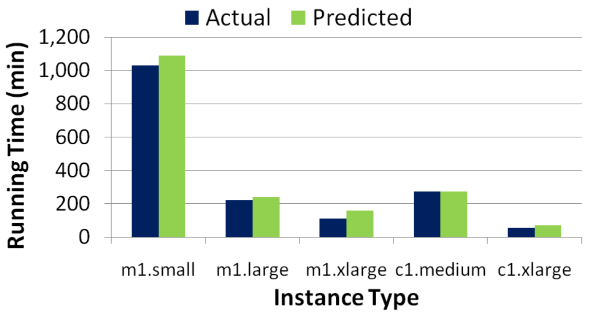
The new and remarkable aspect here is that a nonexpert MapReduce user can provision a cluster of any size on the cloud within minutes to meet her data-processing needs. This feature of the cloud gives tremendous power to the average user, while placing a major burden on her shoulders. Previously, the same user would have had to work with system administrators and management personnel to get a cluster provisioned for her needs. Many days to months would have been needed to complete the provisioning process. Furthermore, making changes to an already-provisioned cluster was a hassle.

Cloud platforms make cluster provisioning almost instantaneous. The elastic and pay-as-you-go nature of these platforms means that, depending on how best to meet her needs, a user can allocate a 10-node cluster today, a 100-node cluster tomorrow, and a 25-node cluster the day after. However, removing the system administrator and the traditional capacity-planning process from the loop shifts the nontrivial responsibility of determining a good cluster configuration to the nonexpert user.

As an illustrative example, consider provisioning a Hadoop cluster on EC2 nodes to run a MapReduce workload on the cloud. Services like Elastic MapReduce and Hadoop On Demand free the user from having to install and maintain the Hadoop cluster. How-

| EC2 Node Type | CPU (# EC2 Units) | Memory (GB) | Storage (GB) | I/O Performance | Cost (U.S. $ per hour) | Map Slots per Node | Reduce Slots per Node | Max Memory per slot (MB) |
|---|---|---|---|---|---|---|---|---|
| m1.small | 1 | 1.7 | 160 | moderate | 0.085 | 2 | 1 | 300 |
| m1.large | 4 | 7.5 | 850 | high | 0.34 | 3 | 2 | 1024 |
| m1.xlarge | 8 | 15 | 1,690 | high | 0.68 | 4 | 4 | 1536 |
| c1.medium | 5 | 1.7 | 350 | moderate | 0.17 | 2 | 2 | 300 |
| c1.xlarge | 20 | 7 | 1,690 | high | 0.68 | 8 | 6 | 400 |

**Table 1: Five representative EC2 node types, along with resources, costs, and cluster parameter settings for Hadoop.**



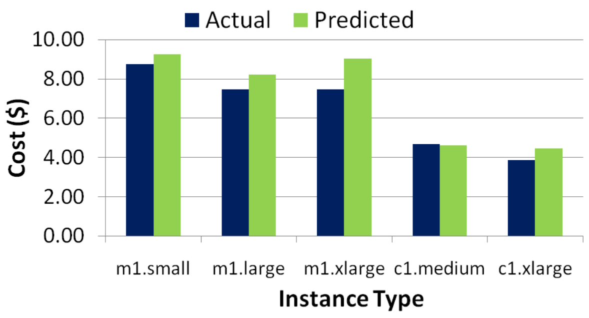


**Figure 1: Performance Vs. pay-as-you-go costs for a workload that is run on different EC2 cluster resource configurations. (a) and (b) show the actual values and the corresponding values as estimated by the Elastisizer.**

ever, the burden of cluster provisioning is still on the user who is likely not an expert system administrator. In particular, the user has to specify the number of EC2 nodes to use in the cluster, as well as the *node type* to use from among 10+ EC2 node types. Table 1 shows the features and renting costs of some representative EC2 node types. Notice that the CPU and I/O resources available on these node types are quoted in abstract terms that an average user will have trouble understanding. To complicate the space of choices even further, the user has to specify what values to use for a number of configuration parameters—e.g., the number of reduce tasks or whether to compress map outputs—at the level of MapReduce job execution on the cluster [2, 14].

In this paper, we refer to the general problem of determining the cluster resources and MapReduce job-level configurations to meet desired requirements on execution time and cost for a given analytic workload as the *cluster sizing problem*. The main contribution of this paper is a system called the *Elastisizer* to which users can express cluster sizing problems as queries. The Elastisizer will provide reliable answers to cluster sizing queries in an automated fashion. In order to illustrate how the Elastisizer benefits users and applications, we begin by discussing some common scenarios where cluster sizing problems arise.

## 1.1 Use Cases for Cluster Sizing Queries

**1. Tuning the cluster size for elastic workloads:** Suppose a MapReduce job takes three hours to finish on a 10-node Hadoop cluster of EC2 nodes of the m1.large type. The application or the user who controls the cluster may want to know by how much the execution time of the job will reduce if five more m1.large nodes are added to the cluster. Such questions arise routinely in practice, and can be answered automatically by the Elastisizer.

**2. Planning for workload transition from a development cluster to production:** Most enterprises maintain separate (and possibly multiple) clusters for application development compared to the production clusters used for running mission-critical and time-sensitive workloads. Elasticity and pay-as-you-go features have simplified the task of maintaining multiple clusters. For example, Facebook uses a *Platinum* cluster that only runs mission-critical

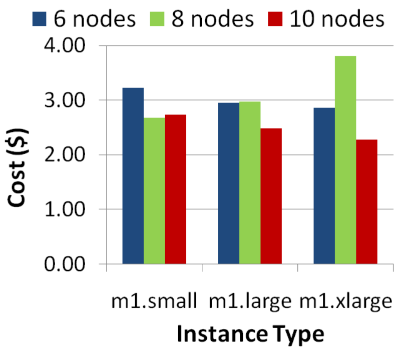


**Figure 2: Pay-as-you-go costs for a workload from Figure 1 when run using auction-based EC2 spot instances.**

jobs [4]. Less critical jobs are run on a separate *Gold* or a *Silver* cluster where data from the production cluster is replicated.

A developer will first test a new MapReduce job on the development cluster, possibly using small representative samples of the data from the production cluster. Before the job can be scheduled on the production cluster—usually as part of an analytic workload that is run periodically on new data—the developer will need to identify a MapReduce job-level configuration that will give good performance for the job when run on the production cluster (without actually running the job on this cluster). The Elastisizer helps the developer with this task. Based on how the job performs when run on the development cluster, the Elastisizer can estimate how the job will run under various hypothetical configurations on the production cluster; and recommend a good configuration to use.

**3. Cluster provisioning under multiple objectives:** Infrastructure-as-a-service (IaaS) cloud platforms like Amazon EC2 and Rackspace offer multiple choices for the type of node to use in a cluster; see Table 1. As the levels of compute, memory, and I/O resources on the nodes increase, so does the cost of renting the nodes per hour. Figure 1 shows the execution time as well as total cost incurred for a MapReduce workload running on Hadoop under different cluster configurations on EC2. The clusters in Figures 1(a) and (b) use six nodes each of the EC2 node type shown, with a fixed per-hour renting cost, denoted *cost_ph* (shown in Table 1). The *pricing model* used to compute the corresponding total cost of each workload execution is:

$$total\_cost = cost\_ph \times num\_nodes \times exec\_time \quad (1)$$

Here, *num_nodes* is the number nodes in the cluster. *exec_time* is the execution time of the workload rounded up to the nearest hour as done on most cloud platforms. The user could have multiple preferences and constraints for the workload. For example, the goal may be to minimize the monetary cost incurred to run the workload, subject to a maximum tolerable workload execution time. Based on Figure 1, if the user wants to minimize cost subject to an execution time of under 45 minutes, then the Elastisizer should recommend a cluster of six c1.xlarge EC2 nodes.

Notice from Figures 1(a) and (b) that the Elastisizer is able to capture the execution trends of the workload correctly across the different clusters. Some interesting tradeoffs between execution time and cost can also be seen in Figure 1. For example, the cluster of six m1.xlarge nodes runs the workload almost 2x faster than the cluster of six c1.medium nodes; but at 1.7x times the cost.

**4. Shifting workloads in time to lower execution costs:** The pricing model from Equation 1 charges a flat per-hour price based on the node type used. Such nodes are called *on-demand instances* on EC2. Amazon EC2 also offers *spot instances* whose prices can vary with time, usually based on the supply and demand for resources on the cloud [8, 12]. Other factors such as temporal and spatial variations in electricity prices can also cause resource usage costs to fluctuate on the cloud [25].

The vast majority of analytics workloads can tolerate some slack in completion time. For example, data analysts in a company may find it sufficient to have an overnight report-generation workload complete before the company's U.S. East Coast offices reopen for business. (Such workloads are called "batch and non-user synchronous workloads" in [12].) This observation gives rise to an online scheduling problem where the slack is exploited to run a given workload when, ideally, resource prices are the cheapest. Solving this online scheduling problem is beyond the scope of this paper since our focus is on the cluster sizing problem. However, the Elastisizer is indispensable in any solution to the scheduling problem since the solution would need estimates of workload execution time and cost for both on-demand and spot instances in various cluster configurations.

As an illustration, Figure 2 shows the total cost incurred for the same MapReduce workload from Figure 1 when nodes of the EC2 spot instance type shown were used to run the workload around 6.00 AM Eastern Time. By comparing Figure 1(b) with Figure 2, it is clear that execution costs for the same workload can differ significantly across different choices for the cluster resources used.

## 1.2 Contributions and Roadmap

Dealing with such use cases for cluster sizing requires considerable manual involvement today, which prevents nonexpert users from taking full advantage of modern cloud platforms. The Elastisizer aims to make timely contributions in this domain. To the best of our knowledge, no other research prototype or commercial system provides the same features as the Elastisizer towards automatically addressing cluster sizing problems for analytic MapReduce workloads on the cloud.

The Elastisizer has been developed as a subsystem of a larger system called *Starfish* [15]. Starfish is a self-tuning system for analytics on large datasets. Starfish builds on Hadoop while adapting to user needs and system workloads to provide good performance automatically, without any need for users to understand and manipulate the many tuning knobs in Hadoop.

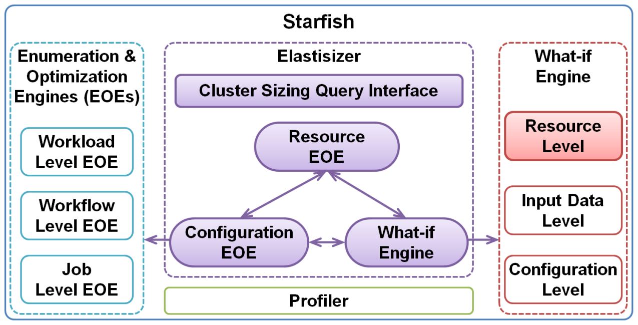


**Figure 3: Elastisizer in the Starfish architecture.**

Figure 3 shows the architecture of the Elastisizer in the context of Starfish. The shaded boxes in Figure 3 represent the new contributions that we have made in order to develop the Elastisizer. The Elastisizer handles cluster sizing problems by making use of a *What-if Engine* and two *Enumeration and Optimization Engines (EOEs)*. The Resource EOE is responsible for the space of possible cluster resources, while the Configuration EOE is responsible for the space of MapReduce job configuration parameter settings.

As indicated in Figure 3, the Elastisizer makes use of some other subsystems in Starfish. The *Profiler* is used to instrument unmodified MapReduce programs dynamically to generate concise statistical summaries, called *job profiles*, of MapReduce job execution. The What-if Engine uses the job profiles to reason about the impact of hypothetical data properties, job configuration settings, and cluster resource properties on MapReduce job execution. Estimating the impact of cluster resource properties on MapReduce job execution is one of the new challenges addressed by the Elastisizer.[1] The What-if Engine is invoked by the EOEs to estimate job performance when the EOEs search through the space of cluster resources and MapReduce job configuration parameter settings to address a given cluster sizing problem.

**Roadmap:** Section 2 discusses related work. Section 3 describes how a user or application expresses a cluster sizing query to the Elastisizer. Section 4 gives an overview of how the Elastisizer answers a cluster sizing query, and Sections 5–7 describe the details of each step in this process. Section 8 presents the results from a comprehensive experimental evaluation of the Elastisizer. We conclude with a discussion in Section 9.

## 2. RELATED WORK

Cluster sizing queries are not unique to cloud platforms. However, what makes the cloud unique in this respect is that cluster sizing problems arise much more frequently here due to the pay-as-you-go and elastic nature. Furthermore, cloud platforms eliminate the traditional system administrator as a necessary middleman between the end-user and the provisioning of cluster resources.

These factors motivate automated approaches to answering cluster sizing queries more strongly than ever. Such an approach has to deal with performance predictions in a complex space of workloads, data properties, cluster resources, configuration settings, and task scheduling policies. The main technical contribution of the Elastisizer is in addressing this complex space using a carefully-designed combination of profiling, black-box models, white-box models, and simulation.

[1]As indicated in Figure 3, the What-if Engine described in this paper is a significantly enhanced version of the What-if Engine described in [14]. In particular, the What-if Engine described in [14] did not have the ability to estimate the impact of changes in cluster resource properties; see Section 6.1.

The Elastisizer addresses the complexities of MapReduce workloads by abstracting them in the form of profiles that can be measured or estimated. The *Profiler* subsystem of Starfish—discussed in detail in [14]—is used to collect a job profile by measurement during the execution of a MapReduce job. The Elastisizer requires measured profiles to be given as input, and it then uses *relative models* aggressively to estimate *virtual profiles* for the workloads when run on hypothetical cluster resources, input data, or job configurations. The complexity of modeling is reduced by considering the discretized space of resource choices (e.g., EC2 node types) offered by IaaS cloud platforms. Simulations using the virtual profiles are used to capture the impact of scheduling policies.

White-box models are created by human experts who have detailed knowledge of how workload execution is affected by the properties of cluster resources, input data, system configuration, and scheduling policies. If the models are developed correctly, then the predictions from them will be very accurate. However, white-box models can quickly become inaccurate if they have to capture the impact of evolving features such as hardware properties.

There has been considerable interest recently in using black-box models like regression trees to build workload performance predictors [3]. These models can be trained automatically from samples of system behavior, and retrained when major changes happen. However, these models are only as good as the predictive behavior of the independent variables they use and how well the training samples cover the prediction space. As the number of independent variables that affect workload performance increases (e.g., data properties, configuration parameter settings, and scheduling policies), the number of training samples needed to learn effective black-box models increases dramatically.

Relative (fitness) modeling is a black-box approach proposed recently for complex storage devices [21]. Relative models are trained to predict how the performance of workloads will change from one device $A$ to another device $B$. Training samples are generated by running a selected benchmark of workloads on both $A$ and $B$. These samples are used to learn a relative $M_{A \to B}$ model that can predict how a workload $W$ will perform on $B$ given the performance of $W$ observed on $A$. Relative models drastically simplify the model-learning process with the caveat that actual workload performance on one of the devices has to be measured before predictions can be made. The Elastisizer borrows ideas from relative modeling and applies them to MapReduce clusters. For example, $A$ and $B$ could respectively be the development and production clusters for use case 2 in Section 1.1.

There have been recent proposals to eliminate modeling altogether, relying instead on actual performance measurements through planned *experiments* [11, 29]. While this promising approach can give accurate predictions for some specific problems, representative experiments are nontrivial to set up and take time to run. Simulation is often a faster alternative to experiments, but developing accurate simulators faces many of the same challenges as analytical white-box models. Some discrete event simulators have been proposed for Hadoop—e.g., *Mumak* [22] and *MRPerf* [26]—but none of them meet the needs of the Elastisizer. Mumak needs a job execution trace as input, and cannot simulate job execution for cluster sizes or job configurations that are different from those used during trace collection. MRPerf needs many minutes per simulation, which makes it very inefficient to address use cases like 2 and 3 from Section 1.1 that need 100-1000s of simulations done over different configuration parameter settings.

Our work shares some goals with a recent work on provisioning Hadoop on cloud platforms [18]. The preliminary approach proposed in [18] uses the following steps: (i) for a training workload of MapReduce jobs, perform brute-force search over the resource configuration space to find the best configuration; (ii) use the collected data to build a signature database that maps resource utilization signatures from the jobs to the optimal configuration; and (iii) given a new job $j$, run a scaled-down version of $j$ to get $j$'s resource utilization signature, and probe the signature database to find the best match. Only two configuration parameters were considered in [18], and no solution was proposed for finding the number of nodes in the cluster. Furthermore, a brute-force approach will not scale to the complex configuration space considered by our Elastisizer.

Given the growing number of commercial cloud platforms, recent research has looked into benchmarking them [19]. Such benchmarks complement our work on building relative models that can predict the performance of a workload $W$ on one provider $A$ based on the performance of $W$ measured on another provider $B$.

## 3. ABSTRACTIONS AND QUERY INTERFACE

A MapReduce workload consists of MapReduce jobs of the form $j = \langle p, d, r, c \rangle$. Here, $p$ represents the MapReduce program that runs as part of the job. $d$, $r$, and $c$ represent respectively the input data, cluster resources, and configuration parameter settings used by the job. In this section, we first present how the Elastisizer captures and represents the properties of $p$, $d$, $r$, and $c$. We will then describe how cluster sizing queries are expressed in a declarative fashion for the Elastisizer to process.

### 3.1 MapReduce Jobs

**Program:** A program $p$ in a MapReduce job $j = \langle p, d, r, c \rangle$ expresses a computation over input data $d$ through two functions: $\textit{map}(k_1, v_1)$ and $\textit{reduce}(k_2, \textit{list}(v_2))$. The $\textit{map}(k_1, v_1)$ function is invoked for every *key-value* pair $\langle k_1, v_1 \rangle$ in the input data $d$; to output zero or more key-value pairs of the form $\langle k_2, v_2 \rangle$. The $\textit{reduce}(k_2, \textit{list}(v_2))$ function is invoked for every unique key $k_2$ and corresponding values $\textit{list}(v_2)$ in the map output; to output zero or more key-value pairs of the form $\langle k_3, v_3 \rangle$.

The map and reduce functions in a given program may be expressed in one among a variety of programming languages like Java, C++, Python, or Ruby. The MapReduce programming model also allows other functions like: (i) $\textit{partition}(k_2)$, for controlling how the map output key-value pairs are partitioned among the reduce tasks, and (ii) $\textit{combine}(k_2, \textit{list}(v_2))$, for performing partial aggregation on the map side. The keys $k_1$, $k_2$, and $k_3$ as well as the values $v_1$, $v_2$, and $v_3$ can be of different and arbitrary types.

**Data:** The properties used by the Elastisizer to represent the input data $d$ in a MapReduce job include $d$'s size, the block layout of files that comprise $d$ in the distributed file-system, and whether $d$ is stored compressed or not. Since the MapReduce methodology is to interpret data (lazily) at processing time, and not (eagerly) at loading time, other properties such as the schema and data-level distributions of $d$ are unavailable by default.

**Cluster resources:** The properties used to represent the cluster resources $r$ include the number of nodes in the cluster, a label per node representing the node type, the cluster's network topology, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot.

**Configuration parameter settings:** Figure 4 illustrates the execution of a MapReduce job $j = \langle p, d, r, c \rangle$. A number of choices have to be made in order to fully specify how the job should execute. These choices, represented by $c$ in $\langle p, d, r, c \rangle$, come from a high-dimensional space of *configuration parameter settings* that includes (but is not limited to):

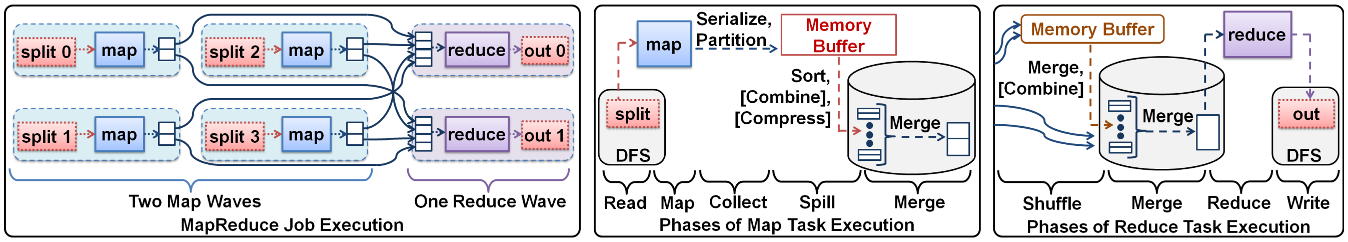


**Figure 4: (a) Execution of a MapReduce job with 4 map tasks (executing in 2 waves) and 2 reduce tasks, (b) zoomed-in version of a map task execution showing the map-side phases, (c) zoomed-in version of a reduce task execution showing the reduce-side phases.**

1. The number of map tasks in job $j$. Each task processes one partition (*split*) of the input data $d$. These tasks may run in multiple *waves* depending on the total number of map task execution slots in $r$.
2. The number of reduce tasks in $j$ (which may also run in waves).
3. The amount of memory to allocate to each map (reduce) task to buffer its outputs (inputs).
4. The settings for multiphase external sorting used by most MapReduce frameworks to group map-output values by key.
5. Whether the output data from the map (reduce) tasks should be compressed before being written to disk (and if so, then how).
6. Whether the program-specified combine function should be used to preaggregate map outputs before their transfer to reduce tasks.

Hadoop has more than 190 configuration parameters out of which the Elastisizer currently considers 14 parameters whose settings can have significant impact on job performance [15].

## 3.2 Profile and Performance of a MapReduce Job Execution

One of the goals of the Elastisizer, and the Starfish system in general [15], is to support the wide and growing variety of MapReduce programs and the programming languages in which they are expressed. Thus, the Elastisizer does not depend on static analysis of MapReduce programs. Instead, a MapReduce program $p$ is represented using one or more profiles that correspond to the execution of $p$ as MapReduce jobs. A profile used for $p$ by the Elastisizer corresponds to the execution of $p$ as a job $j = \langle p, d, r, c\rangle$ over some input data $d$, cluster resources $r$, and configuration parameter settings $c$. This profile is a concise summary of the dataflow and cost information for job $j$'s execution (discussed in Section 5).

A profile is generated by one of two methods. The first method is by direct measurement when job $j$ is run. Section 5.2 describes how profiles are generated by measurement. The second method is by estimation, which does not require the job to be run. Profiles generated by this method are called *virtual profiles*. Section 6 describes how virtual profiles are estimated as part of the cluster sizing process.

The performance of a job $j = \langle p, d, r, c\rangle$ is some function of the properties of $p$, $d$, $r$, and $c$:

$$perf = F(p, d, r, c) \tag{2}$$

Here, *perf* is some performance metric of interest for analytic workloads. The two performance metrics currently supported by the Elastisizer are execution time and execution cost. The pricing model used by the cloud platform has to be known in order to estimate execution costs.

## 3.3 Interface to Express Cluster Sizing Queries

A general cluster sizing problem involves determining the cluster resources and MapReduce job-level configurations to meet desired performance requirements on execution time and cost for a given analytic workload. The Elastisizer provides a declarative interface to express a range of cluster sizing queries including those arising in the four use cases discussed in Section 1.1. Applications and users can also interact with this interface using a programmatic API or using a graphical interface that forms part of the Starfish system's Visualizer [15].

A cluster sizing query specified using the Elastisizer's declarative query interface has four parts. We discuss each of these parts in turn.

**1. Specifying the workload:** The workload specified in a cluster sizing query consists of MapReduce jobs, including both single jobs as well as jobs from multi-job workflows. Each job $j$ runs some MapReduce program $p$ on input data $d$. A profile *prof*$(p)$ has to be given for $p$. If $d$ is different from the input data used while generating *prof*$(p)$, then the properties of $d$ have to be given. For user convenience, when $d$ is actual data on a live cluster, the Elastisizer can collect the properties of $d$ (and $r$) automatically from the cluster. The cluster resources $r$ and job configuration $c$ to use in the actual execution of $j$ are part of separate specifications that involve search spaces, discussed next.

**2. Specifying the search space for cluster resources $r$:** Recall from Section 3.1 that the properties used to represent any cluster resources $r$ include the number of nodes, node type(s), and network topology of $r$, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot. A search space over the number of nodes and the node type to use in the cluster is specified for $r$. This search space is specified as a nonempty set. It is the responsibility of the Elastisizer to find a suitable cluster resource configuration from this search space that meets all other requirements in the query.

The search space for $r$ will be a singleton set if the user is asking for performance estimates for a specific (hypothetical or real) cluster that she has in mind. Use cases 1 and 2 from Section 1.1 have this nature. For example, in use case 1, the user currently has a cluster containing 10 EC2 nodes of the m1.large type. She wants the Elastisizer to estimate what the job's execution time will be on a hypothetical cluster containing 15 nodes of the same type.

The search space is nonsingleton—i.e., it specifies a space that contains more than one candidate resource configuration—when the user wants the Elastisizer to search for a good configuration that meets her needs. In such cases, the Elastisizer's declarative query interface gives the user considerable flexibility to define the search space for cluster resources $r$. For example:

- A unsophisticated user can use the special "*" symbol to ask the Elastisizer to use its default search space for one or both of the node type and the number of nodes.
- The user can restrict the search space to nodes of one or more selected types, but ask the Elastisizer to find the best number of nodes in the cluster.

Our current implementation of the Resource Enumeration and Optimization Engine (see Resource EOE in Figure 3) in the Elastisizer does not include the cluster's network topology as part of the search space for cluster resources. This limitation, which can be removed in future, is driven partly by a practical consideration: most current cloud providers hide the underlying network topology from clients. The Hadoop clusters that we run on EC2 are configured as per the single-rack network topology used as the default in Hadoop.

Note that the current cluster sizing query interface does not expose the other cluster configuration parameters—namely, the number of map and reduce task execution slots per node, and the maximum memory available per task execution slot—as part of the user-specified search space for $r$. Our empirical studies indicate that good settings for these parameters are determined primarily by the CPU (number of cores) and memory resources available per node in the cluster; so we use empirically-determined fixed values per node type (these values are shown in Table 1).[2]

**3. Specifying the search space for job configurations $c$:** Recall the space of configuration parameter settings for MapReduce jobs discussed in Section 3.1. A cluster sizing query needs to specify the search space for configuration parameters $c$ that the Elastisizer should consider for the given workload of MapReduce jobs. Similar to the search space for cluster resources, the search space for $c$ will be a singleton set if the user is asking for performance estimates for a specific configuration that she has in mind. Use case 1 from Section 1.1 has this nature.

It is much more common to specify a larger search space for $c$. The best setting of configuration parameters depends strongly on the cluster resource configuration. For the convenience of nonexpert users who often have no idea about the configuration parameters, the special "*" symbol can be specified to ask the Elastisizer to use its default search space for $c$.

The Configuration Enumeration and Optimization Engine (see Configuration EOE in Figure 3) in the Elastisizer is responsible for searching efficiently through the specified space of configuration parameter settings. The Elastisizer uses the Resource EOE and the Configuration EOE to enumerate and optimize the cluster resources and job configuration parameter settings in tandem. Note that the best job configuration will invariably change if the cluster resources change. In this way, the Elastisizer extends our previous work on optimizing configuration parameter settings for MapReduce jobs for a given cluster [14].

**4. Specifying performance requirements:** Execution time and cost are the two performance metrics supported by the Elastisizer. As part of a cluster sizing query, a user can choose to:

- Have estimated values output for one or both of these metrics.
- Optimize one of these metrics, possibly subject to a constraint on the other metric. For example, optimizing execution cost subject to an execution time under 30 minutes.

## 4. OVERVIEW OF HOW THE ELASTISIZER ANSWERS A CLUSTER SIZING QUERY

This section gives an overview of how the Elastisizer answers a cluster sizing query posed by a user or application. Sections 5–7 will describe the individual steps in more detail.

Consider use case 1 from Section 1.1. A user can express this use case as a cluster sizing query $Q_1$ using the Elastisizer's query interface described in Section 3.3. The user will specify:

- The profile for the three-hour run of the job on 10 m1.large EC2 nodes. Section 5 describes how the profile can be generated by measurement as part of an actual job execution.
- The search space for cluster resources $r$ is a singleton set that specifies 15 EC2 nodes of the (same) m1.large type.
- The search space for the configuration parameter settings $c$ is also a singleton set that specifies the same job configuration as for the 10 node cluster.
- The performance metric of interest is execution time.

The above specification of query $Q_1$ gives the Elastisizer a profile for a job $j = \langle p, d, r_1, c\rangle$. The desired answer is the estimate of execution time for a hypothetical job $j' = \langle p, d, r_2, c\rangle$. Job $j'$ runs the same program $p$ on the same data $d$ and MapReduce configuration $c$ as job $j$. However, the cluster resources used are different between $j$ and $j'$, i.e., $r_1 \neq r_2$.

Query $Q_1$ maps directly to a *what-if question*. The What-if Engine used by the Elastisizer can answer any what-if question of the following general form:

> Given the profile of a job $j = \langle p, d_1, r_1, c_1\rangle$ that runs a MapReduce program $p$ over input data $d_1$ and cluster resources $r_1$ using configuration $c_1$, what will the performance of program $p$ be if $p$ is run over input data $d_2$ and cluster resources $r_2$ using configuration $c_2$? That is, how will job $j' = \langle p, d_2, r_2, c_2\rangle$ perform?

The What-if Engine uses the following two steps to answer a given what-if question (note that job $j'$ is never run in these steps):

1. The first step is to estimate a virtual job profile for the hypothetical job $j'$. This step uses a careful mix of white-box and black-box models. An important contribution we make in this step is how to estimate the virtual profile as the type of node in the cluster, and hence the resource availability per node, is changed (recall Figure 1). Section 6 gives the details of this step. Figure 5 provides an illustration.
2. The virtual profile is then used in a simulation step to estimate how the hypothetical job $j'$ will execute. The answer to the what-if question is computed based on the estimated execution. Section 7 gives the details of this step.

A what-if question can involve multiple jobs in a workload. In this case, all the virtual job profiles are generated, and then input to the simulation step.

Now consider use case 2 from Section 1.1. To express this use case as a cluster sizing query $Q_2$, a user will specify the following:

1. A job profile collected on the development cluster for the program $p$ of interest. This profile may have been collected by direct measurement when $p$ was run on a (scaled down) sample of the production data on which the program has to be run on the production cluster. In this case, the properties of the production data $d_{prod}$ will have to be provided.
2. The search space for cluster resources $r$ is a singleton set that gives the properties of the production cluster, denoted $r_{prod}$.
3. Note that the developer wants the Elastisizer to find the best job configuration to run the job on the production cluster. Thus, the search space for the configuration parameter settings $c$ is specified as "*" so that the Elastisizer will consider the full space in an efficient fashion.
4. The performance metric of interest is execution time.

To process query $Q_2$, the Elastisizer will invoke the Configuration EOE to efficiently enumerate and search through the high-dimensional space of configuration parameter settings. The Configuration EOE will consider hypothetical configurations $c^{(1)}$, $c^{(2)}$, … $c^{(i)}$, …, $c^{(n)}$, making calls to the What-if Engine to get the estimated execution time of each of the corresponding hypothetical jobs $j^{(i)} = \langle p, d_{prod}, r_{prod}, c^{(i)}\rangle$. The configuration $c^{(opt)}$ found in this process with the least execution time will be output as the

[2]Anecdotal evidence from the industry suggests that memory-related misconfigurations are a major cause of failures in Hadoop.

| Profile Field (All fields, unless otherwise stated, represent information at the level of tasks) | Depends On | | |
|---|---|---|---|
| | $d$ | $r$ | $c$ |
| Setup phase time in a task | ✓ | ✓ | ✓ |
| Cleanup phase time in a task | ✓ | ✓ | ✓ |
| Read phase time in the map task | ✓ | ✓ | ✓ |
| Map phase time in the map task | ✓ | ✓ | ✓ |
| Collect phase time in the map task | ✓ | ✓ | ✓ |
| Spill phase time in the map task | ✓ | ✓ | ✓ |
| Merge phase time in map/reduce task | ✓ | ✓ | ✓ |
| Shuffle phase time in the reduce task | ✓ | ✓ | ✓ |
| Reduce phase time in the reduce task | ✓ | ✓ | ✓ |
| Write phase time in the reduce task | ✓ | ✓ | ✓ |

**Table 2: Cost fields in the job profile.**

| Profile Field (All fields, unless otherwise stated, represent information at the level of tasks) | Depends On | | |
|---|---|---|---|
| | $d$ | $r$ | $c$ |
| I/O cost for reading from HDFS per byte | | ✓ | |
| I/O cost for writing to HDFS per byte | | ✓ | |
| I/O cost for reading from local disk per byte | | ✓ | |
| I/O cost for writing to local disk per byte | | ✓ | |
| Cost for network transfers per byte | | ✓ | |
| CPU cost for executing the Mapper per record | | ✓ | |
| CPU cost for executing the Reducer per record | | ✓ | |
| CPU cost for executing the Combiner per record | | ✓ | |
| CPU cost for partitioning per record | | ✓ | |
| CPU cost for serializing/deserializing per record | | ✓ | |
| CPU cost for sorting per record | | ✓ | |
| CPU cost for merging per record | | ✓ | |
| CPU cost for uncompressing the input per byte | | ✓ | |
| CPU cost for uncompressing map output per byte | | ✓ | ✓ |
| CPU cost for compressing map output per byte | | ✓ | ✓ |
| CPU cost for compressing the output per byte | | ✓ | ✓ |
| CPU cost of setting up a task | | ✓ | |
| CPU cost of cleaning up a task | | ✓ | |

**Table 3: Cost statistics fields in the job profile.**

query result; the details of the efficient search process are given in Section 7.

Finally, consider use case 3 from Section 1.1. This use case differs from the earlier use cases in two main ways. First, the search space for cluster resources is not a singleton set any more. Second, the performance requirements demand optimization of execution cost, while specifying a constraint on the execution time.

The Resource EOE and the Configuration EOE will be used in tandem to enumerate and search over the space of cluster resources and job configurations; while making calls to the What-if Engine to get estimates of execution time and cost for hypothetical jobs $j^{(i,j)}$ = $\langle p, d, r^{(i)}, c^{(j)} \rangle$. The combination $\langle r, c \rangle^{(opt)}$ found in this process that gives the least execution cost while meeting the constraint on execution time will be output as the query result.

## 5. PROFILE OF A MAPREDUCE JOB

A MapReduce job execution consists of the execution of map tasks and reduce tasks. As illustrated in Figure 4, task execution can be broken down further to the level of phases within tasks.

Map task execution is divided into the phases: *Read* (reading map inputs), *Map* (map function processing), *Collect* (buffering map outputs before spilling to local disk), *Spill* (sorting, combining, compressing, and writing map outputs to local disk), and *Merge* (merging sorted spill files). Reduce task execution is divided into the phases: *Shuffle* (transferring map outputs to reduce tasks, with decompression if needed), *Merge* (merging sorted map outputs), *Reduce* (reduce function processing), and *Write* (writing reduce outputs to the distributed file-system). Additionally, both map and reduce tasks have *Setup* and *Cleanup* phases.

| Profile Field (All fields, unless otherwise stated, represent information at the level of tasks) | Depends On | | |
|---|---|---|---|
| | $d$ | $r$ | $c$ |
| Number of map tasks in the job | ✓ | | ✓ |
| Number of reduce tasks in the job | | | ✓ |
| Map input records | ✓ | | ✓ |
| Map input bytes | ✓ | | ✓ |
| Map output records | ✓ | | ✓ |
| Map output bytes | ✓ | | ✓ |
| Number of spills | ✓ | | ✓ |
| Number of merge rounds | ✓ | | ✓ |
| Number of records in buffer per spill | ✓ | | ✓ |
| Buffer size per spill | ✓ | | ✓ |
| Number of records in spill file | ✓ | | ✓ |
| Spill file size | ✓ | | ✓ |
| Shuffle size | ✓ | | ✓ |
| Reduce input groups (unique keys) | ✓ | | ✓ |
| Reduce input records | ✓ | | ✓ |
| Reduce input bytes | ✓ | | ✓ |
| Reduce output records | ✓ | | ✓ |
| Reduce output bytes | ✓ | | ✓ |
| Combiner input records | ✓ | | ✓ |
| Combiner output records | ✓ | | ✓ |
| Total spilled records | ✓ | | ✓ |
| Bytes read from local file system | ✓ | | ✓ |
| Bytes written to local file system | ✓ | | ✓ |
| Bytes read from HDFS | ✓ | | ✓ |
| Bytes written to HDFS | ✓ | | ✓ |

**Table 4: Dataflow fields in the job profile.**

| Profile Field (All fields, unless otherwise stated, represent information at the level of tasks) | Depends On | | |
|---|---|---|---|
| | $d$ | $r$ | $c$ |
| Width of input key-value pairs | ✓ | | |
| Number of records per reducer's group | ✓ | | |
| Map selectivity in terms of size | ✓ | | |
| Map selectivity in terms of records | ✓ | | |
| Reducer selectivity in terms of size | ✓ | | |
| Reducer selectivity in terms of records | ✓ | | |
| Combiner selectivity in terms of size | ✓ | | ✓ |
| Combiner selectivity in terms of records | ✓ | | ✓ |
| Input data compression ratio | ✓ | | |
| Map output compression ratio | ✓ | | ✓ |
| Output compression ratio | ✓ | | ✓ |
| Startup memory per task | ✓ | | |
| Setup memory per task | ✓ | | |
| Memory per map's record | ✓ | | |
| Memory per reducer's record | ✓ | | |
| Cleanup memory per task | ✓ | | |

**Table 5: Dataflow statistics fields in the job profile.**

The profile of a MapReduce job consists of fields that together form a concise summary of the job's execution at the level of tasks and phases. We partition the fields in a profile into four categories, described next. The rationale for this categorization will become clear in Section 6 when we describe how a virtual profile is estimated for a hypothetical job without running the job.

### 5.1 Categorization of the Fields in a Profile

**Cost fields:** These fields capture information about execution time at the level of tasks as well as phases within the tasks for a MapReduce job execution. Table 2 lists all the cost fields in a profile.

**Cost statistics fields:** These fields capture statistical information about execution time for a MapReduce job—e.g., the average time to read a record from the distributed filesystem, or the average time to execute the map function per input record—that is expected to remain unchanged across different executions of the job unless the CPU and/or I/O resources available per node change. Table 3 lists all the cost statistics fields in a profile.

**Dataflow fields:** These fields capture information about the amount of data, both in terms of bytes as well as records (key-value pairs), flowing through the different tasks and phases of a MapReduce job execution. An example field is the number of map output records. Table 4 lists all the dataflow fields in a profile.

**Dataflow statistics fields:** These fields capture statistical information about the dataflow—e.g., the average number of records output by map tasks per input record (the Map selectivity)—that is expected to remain unchanged across different executions of the MapReduce job unless the data distribution in the input dataset changes significantly across these executions. Table 5 lists all the dataflow statistics fields in a profile.

## 5.2 Generating Profiles by Measurement

We now discuss how the Profiler in Starfish (Figure 3) generates a new profile for a job by measurement when the job is run. The Profiler uses *dynamic instrumentation* to collect run-time monitoring information from unmodified MapReduce programs running on the MapReduce framework. Dynamic instrumentation is a technique used to understand, debug, and optimize complex systems [6]. The dynamic nature means that there is zero overhead when monitoring is turned off; an appealing property in production deployments. By supporting unmodified MapReduce programs, we free users from any additional burden on their part for generating profiles to use in cluster sizing queries (recall Section 3.3).

The Profiler specifies a set of event-condition-action rules. The space of events corresponds to events arising during program execution such as entry or exit from specific functions and any system calls to the operating system. If the condition associated with the event holds when the event fires, then the associated action is invoked. An action can involve, for example, getting the duration of a function call, examining the memory state, or logging the event.

Our current implementation of the Profiler for Hadoop uses the *BTrace* dynamic instrumentation tool for Java [5]. Hadoop itself is written in Java, but it can run MapReduce jobs written in many other programming languages like C++, Python, and Ruby. When Hadoop runs any MapReduce program, the Profiler can dynamically instrument selected Java classes internal to Hadoop to collect raw monitoring data. Under the covers, dynamic instrumentation intercepts the class bytecodes at run-time based on the event-condition-action rules (also specified in Java), and injects bytecodes for the associated actions. Raw monitoring data includes record and byte counters, resource usage, and timings. For instance, during each map-side Spill phase (see Figure 4(b)), the exit point of the sort function is instrumented to collect the sort duration as well as the number of records and bytes sorted.

The raw monitoring data collected through dynamic instrumentation for each profiled map or reduce task is first processed to give fields in a *task profile*. For example, the raw sort timings are added as part of the overall spill time, whereas the Combiner selectivity from each spill is averaged to get the task's Combiner selectivity. The map task profiles are further processed to give one representative map task profile for each logical input to the MapReduce program. For example, a Sort program accepts a single logical input (be it a single file, a directory, or a set of files), while a two-way Join accepts two logical inputs. The reduce task profiles are processed into one representative reduce task profile. The map task profile(s) and the reduce task profile together constitute the job profile.

Since the Profiler instruments the MapReduce framework only, not the user-written programs, profiling works irrespective of the programming language in which the program is written. While dynamic instrumentation has zero impact when turned off, it causes some task slowdown when enabled. To keep the additional overhead on job execution as low as possible, the Profiler supports task-level sampling in order to generate approximate job profiles [14].

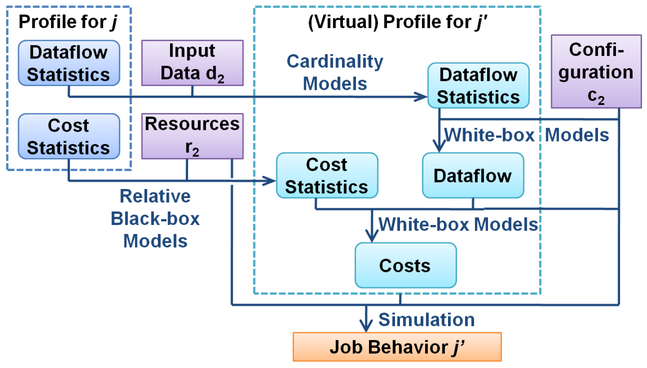


**Figure 5: Overall process for estimating virtual job profiles.**

## 6. ESTIMATING VIRTUAL JOB PROFILES

We will now describe how virtual profiles are estimated by the What-if Engine as part of answering a what-if question. Specifically, given the profile of a job $j = \langle p, d_1, r_1, c_1 \rangle$ and the properties of input data $d_2$, cluster resources $r_2$, and configuration parameter settings $c_2$ of a hypothetical job $j' = \langle p, d_2, r_2, c_2 \rangle$, the virtual profile of $j'$ has to be estimated. This process of virtual profile estimation forms the foundation on which the Elastisizer's ability to answer cluster sizing queries is based.

Our solution for virtual profile estimation is based on a mix of black-box and white-box models. The overall estimation process has been broken down into smaller steps as shown in Figure 5, and a suitable modeling technique was picked for each step. These smaller steps correspond to the four categories of fields in a job profile. We use relative black-box models to estimate the cost statistics fields, conventional white-box models from database query optimization to estimate dataflow statistics fields, and new white-box models that we developed to estimate the dataflow fields, and in turn, the cost fields. We describe each of these in turn.

## 6.1 Relative Black-box Models to Estimate Cost Statistics Fields

Let us first consider the cost statistics fields shown in Table 3. Clusters with identical resources will have the same CPU and I/O (local and remote) costs. Thus, if the cluster resource properties of $r_1$ are the same as those of $r_2$, then the cost statistics fields in the hypothetical job $j'$ virtual profile can be copied directly from the profile of job $j$ given as input. This copying cannot be used when $r_1 \neq r_2$, in particular, when job $j'$ will run on a *target cluster* containing nodes with a different type from the *source cluster* where job $j$ was run to collect the profile that was given as input.

The technique we use when $r_1 \neq r_2$ is based on a *relative* black-box model $M_{src \to tgt}$ that can predict the cost statistics fields $CS_{tgt}$ in the virtual profile for the target cluster from the cost statistics fields $CS_{src}$ in the job profile for the source cluster.

$$CS_{tgt} = M_{src \to tgt}(CS_{src}) \tag{3}$$

**Generating training samples for the $M_{src \to tgt}$ model:** Let $r_{src}$ and $r_{tgt}$ denote the cluster resources respectively for the source and target clusters. Suppose the MapReduce program $p$ is run on input data $d$ and configuration parameter settings $c$ on both the source and target clusters. That is, we run the two jobs $j_{src} = \langle p, d, r_{src}, c \rangle$ and $j_{tgt} = \langle p, d, r_{tgt}, c \rangle$. From these runs, we can generate the profiles for these two jobs by direct measurement. Even further, recall from Section 5.2 that we can generate a separate task profile for each task run in each of these two jobs. Therefore, from the $i^{th}$ task in these two jobs,[3] we get a training sample $\langle \text{CS}_{src}^{(i)}, \text{CS}_{tgt}^{(i)}, \rangle$ for the $M_{src \to tgt}$ model.

The above training samples were generated by running a related pair of jobs $j_{src}$ and $j_{tgt}$ that have the same MapReduce program $p$, input data $d$, and configuration parameter settings $c$. We can generate a complete set of training samples by using a *training benchmark* containing jobs with different $\langle p, d, c \rangle$ combinations. Selecting an appropriate training benchmark is nontrivial because the two main requirements of effective black-box modeling have to be satisfied. First, for accurate prediction, the training samples must have good coverage of the prediction space. Second, for efficiency, the time to generate the complete set of training samples must be small.

Based on the coverage and efficiency considerations, we came up with three methods to select the training benchmark in the Elastisizer: *Apriori*, *Fixed*, and *Custom*. We will evaluate these three benchmarks empirically in Section 8.

*Apriori:* This method assumes that the full workload of jobs that will run on the provisioned clusters is known at model training time. A sample of jobs is selected from this workload, either randomly or from the top-k longest-running jobs. A similar approach of sampling the SQL query workload is used by the index selection and other wizards in most commercial database systems [7]. To improve the training efficiency, it may be possible to run the jobs on a scaled-down sample of their input data. However, this step requires domain knowledge or user assistance. Apriori gives good coverage of the prediction space as long as the assumption on the workload holds. However, Apriori's running time grows with the size of the workload and the input data.

*Fixed:* This method assumes that a good coverage of the prediction space can be obtained by selecting a predetermined set of existing MapReduce jobs (e.g., Sort, WordCount) and executing them using different configuration settings that will give different degrees of resource usage. For example, the benchmark can consist of CPU-intensive, CPU-light, I/O-intensive, and I/O-light jobs. The running time of Fixed is independent of the size of the actual workload and the input data.

*Custom:* The goal of this method is to execute a small, synthetic workload to generate training samples for cost statistics efficiently such that these samples will give good coverage of the prediction space. It is because of our abstraction of any MapReduce job execution as a job profile—where a profile can be represented as a point in a high-dimensional space (see Section 5)—that we are able to consider such a unique approach that is independent of the actual MapReduce workload run on the cluster.

Our Custom training benchmark is composed of just two synthetic MapReduce *job templates*: a data-generation template and a data-processing template. These two templates are instantiated in different ways for a total of six MapReduce job executions. Unlike the Fixed benchmark that consists of existing MapReduce jobs, the jobs generated by Custom have been designed such that the different tasks within these jobs behave differently in terms of their CPU, I/O, memory, and network usage.

While this approach may sound counterintuitive because the map (reduce) tasks in a job are expected to behave similarly, it produces more diverse training samples per job execution than Apriori or Fixed. Custom provides two additional advantages: (i) lower and more predictable running time for the training benchmark; and (ii) no knowledge or use of actual workloads and input data is needed during the training phase.

**Learning all the $M_{src \to tgt}$ models needed:** It is important to note that the training benchmark has to be run only once (or with a few repetitions) per target cluster resource; giving only a linear number of benchmark runs, and not quadratic as one might expect from the relative nature of the $M_{src \to tgt}$ models. The training samples for each source-to-target cluster pair is available from these runs. For example, to address use case 2 from Section 1.1, one run each of the training benchmark on the development and the production cluster will suffice. For a more complex scenario like use case 3 that involves different types of Amazon EC2 nodes, one benchmark run for each distinct node type and a representative number of cluster nodes is usually sufficient. If the workload or data size is expected to vary widely, then benchmark runs over a few different numbers of nodes in the cluster can improve prediction accuracy.

Once the training samples are generated, there are many *supervised learning* techniques available for generating the black-box model in Equation 3. Since cost statistics are real-valued, we selected the *M5 Tree Model* [24]. An M5 Tree Model first builds a regression-tree using a typical decision-tree induction algorithm. Then, the tree goes through pruning and smoothing phases to generate a linear regression model for each leaf of the tree.

## 6.2 White-box Models to Estimate Dataflow Statistics Fields

Database systems keep fine-grained statistics such as histograms for input data. Database query optimizers use these statistics to estimate the dataflow in execution plans for declarative queries. MapReduce frameworks lack the declarative query semantics and structured data representations of database systems. Thus, the common case in the What-if Engine is to not have detailed statistical information about the input data $d_2$ in the hypothetical job $j'$. By default, the What-if Engine makes a *dataflow proportionality assumption* which says that the logical dataflow sizes through the job's phases are proportional to the input data size. It follows from this assumption that the dataflow statistics fields in the virtual profile of $j'$ will be the same as those in the profile of job $j$ given as input.

When additional information is available, the What-if Engine allows the default assumption to be overridden by providing dataflow statistics fields of the virtual profile directly as input. For example, when higher semantic layers like Hive and Pig submit a MapReduce job for a computation like filtering or join, they can estimate dataflow statistics fields like Map and Reduce selectivity using conventional statistics like histograms. Researchers are also developing tools to extract detailed information from MapReduce programs through program analysis [16, 17].

## 6.3 White-box Models to Estimate Dataflow and Cost Fields

The What-if Engine uses a detailed set of analytical (white-box) models to calculate the dataflow fields in the virtual profile given (i) the dataflow statistics fields estimated above, and (ii) the configuration parameter settings $c_2$ in the hypothetical job $j'$. These models give good accuracy by capturing the subtleties of MapReduce job execution at the fine granularity of phases within map and reduce tasks. The current models were developed for Hadoop, but the overall approach applies to any MapReduce implementation. A second set of analytical models combines the estimated cost statistics and dataflow fields in order to estimate the cost fields in the virtual profile. The full set of models is described in a technical report that is available online [13].

[3]Ensuring that the tasks have the same input data $d$ and configuration parameter settings $c$ ensures that there is a one-to-one correspondence between the tasks in these two jobs.

| Abbr. | MapReduce Program | Dataset Description |
|---|---|---|
| LG | LinkGraph | 20GB compressed data from Wikipedia |
| JO | Join | 60GB data from the TPC-H Benchmark |
| TF | TF-IDF | 60GB of documents from Wikipedia |
| TS | Hadoop's TeraSort | 60GB data from Hadoop's TeraGen |
| WC | WordCount | 60GB of documents from Wikipedia |
| CO | Word Co-occurrence | 10GB of documents from Wikipedia |

**Table 6: MapReduce programs and corresponding datasets.**

## 7. ANSWERING CLUSTER SIZING QUERIES

We saw in the previous section how a virtual job profile containing detailed dataflow and cost information at the task and phase level is generated for the hypothetical job $j'$. The What-if Engine then uses a *Task Scheduler Simulator*, along with the models and information on the cluster resources $r_2$, to simulate the scheduling and execution of the map and reduce tasks of $j'$. The output is a description of the complete (hypothetical) job execution in the cluster. The Task Scheduler Simulator is a pluggable component with a current implementation that mimics Hadoop's default scheduler. The answer to the what-if question, e.g., execution time and/or cost, is derived from the job's simulated execution.

As we showed in Section 4, the cluster sizing query $Q_1$ for use case 1 is answered by a single what-if question. More complex queries, e.g., $Q_2$ for use case 2 and $Q_3$ for use case 3, require a search over the space of configuration parameter settings and/or cluster resources. These queries are answered by the EOEs in Figure 3 based on answers to a series of what-if questions generated through a search process called *Recursive Random Search (RRS)*. RRS is a fairly recent technique developed to solve black-box optimization problems [28]. RRS first samples the search space randomly to identify promising regions that contain the optimal setting with high probability. It then samples recursively in these regions which either move or shrink gradually to locally-optimal settings based on the samples collected. RRS then restarts random sampling to find a more promising region to repeat the recursive search. The efficiency of the EOEs comes from how RRS is able to decrease the number of what-if calls by many orders of magnitude compared to exhaustive search, while still finding near-optimal settings [14].

## 8. EXPERIMENTAL EVALUATION

In our experimental evaluation, we used Hadoop clusters running on Amazon EC2 nodes of various sizes and node types. Table 1 lists the EC2 node types we used, along with the resources available for each node type. For each node type, we used empirically-determined fixed values for the cluster-wide Hadoop configuration parameters—namely, the number of map and reduce task execution slots per node, and the maximum memory available per task slot (also shown on Table 1).

Table 6 lists the MapReduce programs and datasets used in our evaluation. We selected representative MapReduce programs used in different domains: text analytics (WordCount), natural language processing (Word Co-occurrence), information retrieval (TF-IDF: Term Frequency-Inverse Document Frequency[4]), creation of large hyperlink graphs (LinkGraph), and business data processing (Join, TeraSort) [20, 27]. The unoptimized MapReduce jobs are executed using rules-of-thumb settings found in [9]. Unless otherwise noted, we used the training samples produced by the Custom benchmark to train the relative models for estimating cost statistics (recall Section 6.1).

The goal of the experimental evaluation is to study the ability of the Elastisizer to provide reliable answers to cluster sizing queries. With this goal, we guide the evaluation using the common scenarios where cluster sizing problems arise (discussed in Section 1.1). Our evaluation methodology is as follows:

- We evaluate the predictive power of the Elastisizer for tuning the cluster size for elastic MapReduce workloads.
- We evaluate the optimization capabilities of the Elastisizer in finding a good configuration setting for a program $p$ to be run on the production cluster ($r_{tgt}$) based on a profile learned for $p$ on the development cluster ($r_{src}$).
- We evaluate the cluster provisioning capabilities of the Elastisizer for a MapReduce workload under the dual objectives of execution time and cost.
- We evaluate the accuracy of the relative models learned for predicting cost statistics for the three training benchmarks developed to generate training samples (Section 6.1).

Since our evaluation concentrates on the Elastisizer, we focus on the job running times and monetary costs, and ignore any data loading times.

[4]TF-IDF is a workload consisting of three MapReduce jobs.

### 8.1 Tuning the Cluster Size

Cluster sizing queries regarding the performance of MapReduce jobs on clusters of different sizes arise routinely in practice. Hence, the query we consider in this section is how will the performance of a MapReduce job change if the number of nodes in the existing cluster changes? We evaluate the ability of the Elastisizer to answer such a query automatically.

Figure 6 shows the actual and predicted running times for all MapReduce jobs as the number of nodes in the cluster is varied. All Hadoop clusters for this experiment used m1.large EC2 nodes. To make the predictions, we used job profiles that were obtained on a 10-node Hadoop cluster of m1.large EC2 nodes. We observe that the Elastisizer is able to capture the execution trends of all jobs across the clusters with different sizes. That is, the Elastisizer predicts correctly the sublinear speedup achieved for each job as we increase the number of nodes in the cluster.

From the perspective of predicting absolute values, the Elastisizer usually over-predicts job execution time (by 20.1% on average and 58.6% worse case in Figure 6). The fairly uniform gap between the actual and predicted execution times is due to overhead added by BTrace while measuring function timings at nanosecond granularities. Since dynamic instrumentation mainly needs additional CPU cycles, the gap is largest when the MapReduce job runs under CPU contention (which is the case for TF-IDF and WordCount in Figure 6). While the gap is fairly uniform for different settings of the same MapReduce job, the gap among different jobs varies significantly, making it difficult to correct for it during the prediction process. We are hoping to close this gap using commercial Java profilers that have demonstrated vastly lower overheads than BTrace [23].

Because of its uniformity, the gap does not affect the accuracy of optimization and provisioning decisions which are more about predicting relative changes in performance correctly rather than predicting absolute performance. Therefore, despite the gap, the Elastisizer is able to find the best resource and configuration settings, as we will see in Sections 8.2 and 8.3.

### 8.2 Transitioning from Development to Production

The second common use case we consider in our evaluation is the presence of a development cluster, and the need to stage jobs from the development cluster to the production cluster. In our evaluation, we used a 10-node Hadoop cluster with m1.large EC2 nodes as the development cluster, and a 30-node Hadoop cluster with m1.xlarge

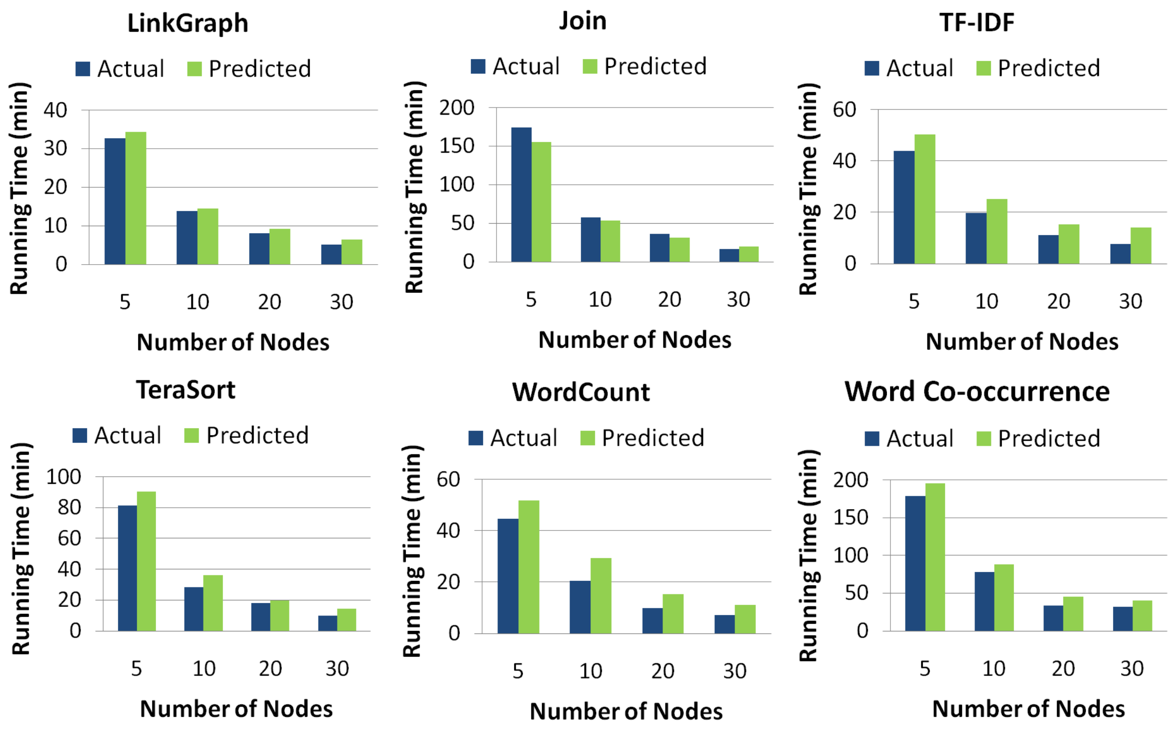


**Figure 6: Actual and predicted running times for MapReduce jobs as the number of nodes in the cluster is varied.**

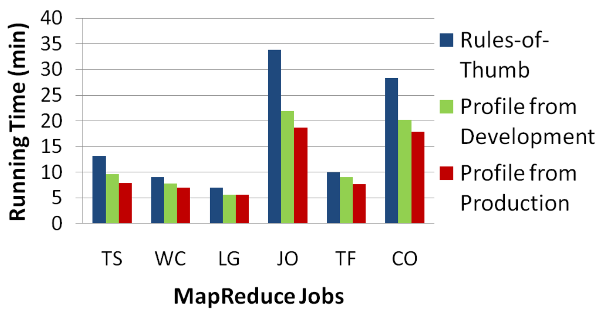


**Figure 7: Job execution times when run with rules-of-thumb settings, and Elastisizer-suggested settings using job profiles obtained from the development and production clusters.**

EC2 nodes as the production one. We profiled all MapReduce programs listed in Table 6 on the development cluster. We then executed the MapReduce programs on the production cluster using three times as much data as used in the development cluster (i.e., three times as much data as listed in Table 6).

Figure 7 shows the running time for each MapReduce job when run on the production cluster with the Elastisizer-suggested configuration settings. In all cases, the suggested settings improved the job performance by an average speedup of 1.4x and a maximum speedup of 1.8x, compared to the rules-of-thumb settings obtained from manually tuning each job. For comparison purposes, we also profiled the MapReduce jobs when run on the production cluster, and then asked the Elastisizer for the best configuration settings. We observe from Figure 7 that, in most cases, the performance improvement achieved over the rules-of-thumb settings is almost the same, irrespective of whether the Elastisizer used the job profiles from the development cluster or the production cluster.

Apart from providing good configuration settings with which to run the jobs on the production cluster, the Elastisizer will also predict the job behavior. Figure 8 shows the actual and predicted running times for each job when run with the settings suggested by the Elastisizer. Apart from the overall running time, the Elastisizer can also predict several other aspects of the job execution like the amount of I/O and network traffic, the running time and scheduling of individual tasks, and data and computational skew.

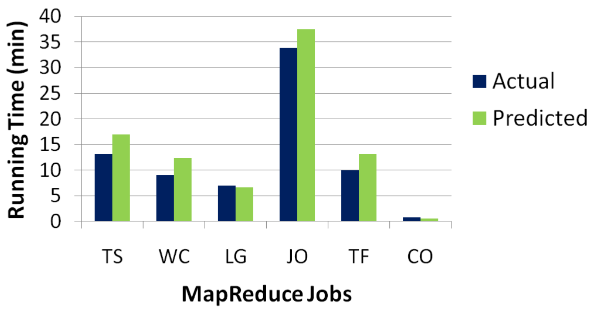


**Figure 8: Actual and predicted running times for MapReduce jobs when run on the production cluster. The predictions used job profiles obtained from the development cluster.**

## 8.3 Multi-objective Cluster Provisioning

As discussed in Section 7, the Elastisizer can reason about the Hadoop job parameter configuration space as well as about the resource space. The resource space includes the type of node and the number of nodes in a cluster. For our evaluation, we used the 5 EC2 node types listed in Table 1. In this section, we evaluate the ability of the Elastisizer to find good cluster and job configuration settings to use for a MapReduce workload.

The workload we used consists of the MapReduce jobs listed in Table 6 run one after the other. The job profiles were obtained by running the workload on a 10-node cluster of m1.large EC2 nodes. For cost predictions, the Elastisizer uses a pricing model containing the hourly node costs listed in Table 1.

Figure 9 shows the running time of the workload when run with the configuration settings suggested by the Elastisizer, across clus-

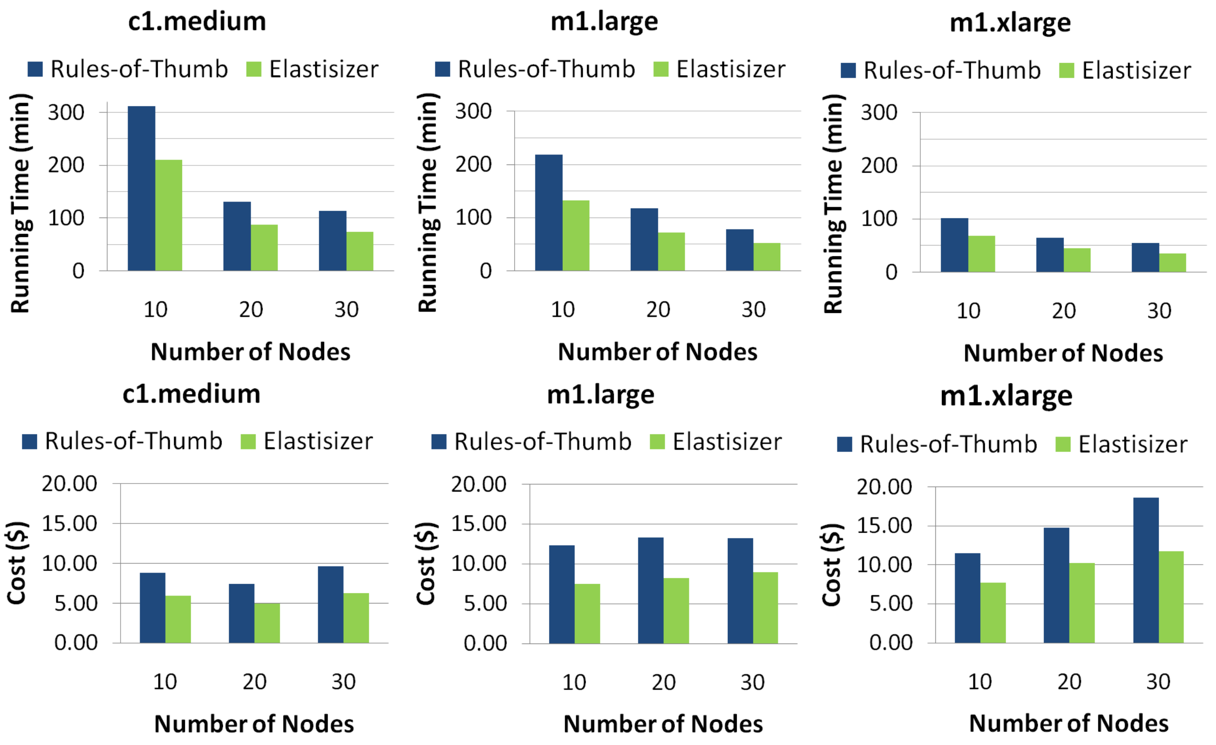


**Figure 9: Running time and monetary cost of the workload when run with (a) rules-of-thumb settings and (b) Elastisizer-suggested settings, while varying the number of nodes and node types in the clusters.**

ters each with a different type of node and number of nodes. The Elastisizer was able to provide up to 1.7x speedup for the workload, which translates into 42% cost savings. Since the Elastisizer is able to reason about the combined resource and configuration space accurately, the Elastisizer is also able to answer general provisioning queries of the form: "What is the best combination of cluster resources and configuration settings to use for running my workload in a way that minimizes execution time (or monetary cost), subject to a maximum tolerable monetary cost (or execution time)?" In our experiment, the Elastisizer was able to identify correctly that using a 30-node cluster with m1.xlarge nodes would yield the best workload execution time, whereas using a 20-node cluster with c1.medium nodes would minimize the monetary cost.

It is interesting to note the complex interactions between execution times and monetary costs as we vary the number of nodes and node type used in the clusters. As expected, increasing the number of nodes and using more powerful machines lead to better running times. However, the performance improvements are not necessarily linear. Let us consider the 10-node cluster with m1.xlarge nodes. If we use 3x more nodes, then we achieve only 2x performance improvement for the workload, but for only a 1.5x increase in cost. On the other hand, the same 3x increase in the number of nodes for a cluster with m1.large nodes leads to an almost 3x performance improvement with only a 1.2x increase in cost.

The Elastisizer is able to capture these complex interactions with a good degree of accuracy; so it can help users select the best cluster resources to fit their needs and preferences.

## 8.4 Evaluation of Training Benchmarks

The ability of the Elastisizer to make accurate predictions across clusters relies on the relative models employed to predict cost statistics. The models we used, like all black-box models, require representative training data in order to make accurate predictions. As discussed in Section 6, we developed three training benchmarks that employ different strategies to collect training samples.

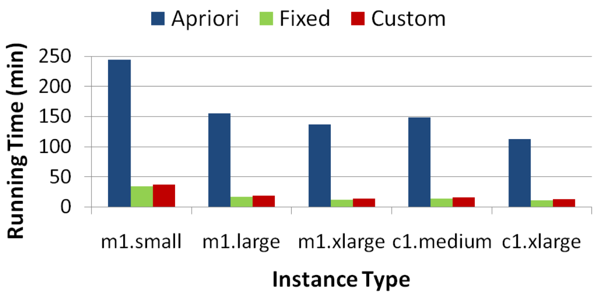


**Figure 10: Total running time for each training benchmark.**

**Apriori benchmark:** This benchmark includes all jobs listed in Table 6, which also form our testing workload. Each job runs over a 600MB random sample of the original input data.

**Fixed benchmark:** This benchmarks executes the MapReduce jobs WordCount and TeraSort multiple times using different configuration settings. We varied the settings for using intermediate data compression, output compression, and the combiner function, since these settings provide tradeoffs between CPU and I/O usage. Each job processed 600MB of randomly generated text using Hadoop's RandomTextGenerator and TeraGen.

**Custom benchmark:** This benchmark consists of a data generation job template and a data processing job template as discussed in Section 6. The data generation job template is run twice (with output compression turned on and off) and the data processing job template is run four times (corresponding to the four possible combinations of using compression and combiner).

All benchmarks were run on 10-node Hadoop clusters on EC2 nodes. Each job in each benchmark processed 600MB of data and was run using rules-of-thumb settings. We tested the prediction

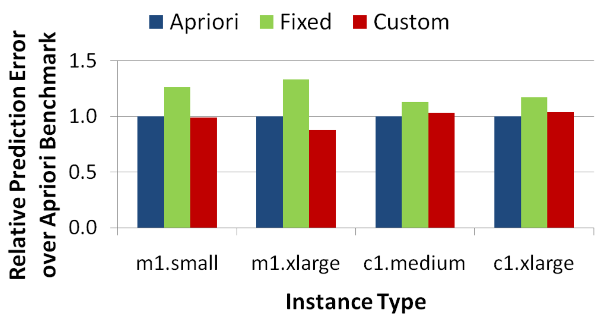


**Figure 11: Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark when asked to predict cost statistics for a test workload.**

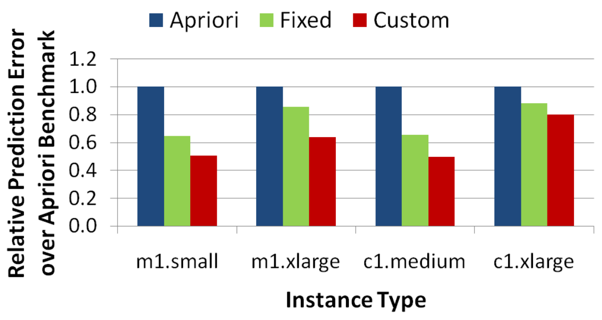


**Figure 12: Relative prediction error for the Fixed and Custom benchmarks over the Apriori benchmark without TF-IDF when asked to predict cost statistics for the TF-IDF job.**

accuracy of the relative models trained by each benchmark on a test workload consisting of all jobs listed in Table 6.

Figure 10 shows the running time of each benchmark for collecting all the training data. The Apriori benchmark takes a significantly longer time to complete compared to the other two benchmarks as it executes more MapReduce jobs. The Custom benchmark, on the other hand, completes fast due to its focused nature of going after a spectrum of cost statistics within the same job.

In order to compare the prediction accuracy of the relative models when trained with the three benchmarks, we created a test workload consisting of all MapReduce jobs from Table 6. The test workload was executed on five 10-node Hadoop clusters—one for each node type we considered in our evaluation (see Table 1). We then used the job profiles obtained on the m1.large cluster to predict the job profiles for the other four clusters (i.e., relative predictions). As the Apriori benchmark assumes full knowledge of the test workload, we will use it as the baseline when comparing the prediction accuracy of the three benchmarks.

Figure 11 shows the relative prediction error from using the Fixed and Custom benchmarks against using the Apriori benchmark. Even though the processing performed by the jobs in the Custom benchmark is completely independent from and unrelated to the test workload, the prediction errors we observed are relatively low, typically less than 15%. The Fixed benchmark results in the highest prediction errors: running a predefined set of jobs with various settings does not seem to provide adequate coverage of the possible cost statistics encountered during the execution of the test workload.

Even though the Apriori benchmark leads to good predictions when the test workload contains the same or similar jobs with the training workload, it can lead to poor predictions for new jobs. For evaluation purposes, we excluded the TF-IDF job from the training workload of the Apriori benchmark. We then tested the relative models with the TF-IDF job profiles. We observed higher prediction errors compared to predictions for the other jobs: Figure 12 shows how the Apriori benchmark is now outperformed by the Custom benchmark.

Overall, when the workload is known a priori and the high time to collect the training data is not a concern, using the Apriori benchmark is the best option. Otherwise, the Custom benchmark is a reliable and robust option for collecting training samples quickly that lead to good predictions.

## 8.5 Approximate Profiles through Sampling

Profiling, when enabled, causes some slowdown in the running time of a MapReduce job $j$. To minimize this overhead, the Profiler can selectively profile a random fraction of the tasks in $j$. For this experiment, we used a 16-node cluster with c1.medium nodes to profile Word Co-occurrence (a CPU-intensive job) and TeraSort (an I/O-intensive job) while enabling profiling for only a sample of the tasks in each job. As we vary the percentage of profiled tasks in each job, Figure 13(a) shows the profiling overhead by comparing against the same job run with profiling turned off. For both MapReduce jobs, as the percentage of profiled tasks increases, the overhead added to the job's running time also increases (as expected). It is interesting to note that the profiling overhead varies significantly between the two jobs. The magnitude of the profiling overhead depends on whether the job is CPU-bound, uses a combiner, uses compression, as well as the job configuration settings.

Figure 13(b) shows the speedup achieved in job execution time by the Elastisizer-suggested settings over the rules-of-thumb settings as the percentage of profiled tasks used to generate the job profile is varied. In most cases, the settings suggested by the Elastisizer led to nearly the same job performance improvements; showing that the Elastisizer's effectiveness in finding good configuration settings does not require that all tasks be profiled. Therefore, by profiling only a small fraction of the tasks in a job, we can keep the overhead low while achieving high degrees of accuracy in the information collected.

## 9. DISCUSSION AND FUTURE WORK

IaaS cloud platforms allow nonexpert users to provision clusters of any size on the cloud to run their MapReduce workloads, and pay only for the resources used. However, these users are now faced with complex cluster sizing problems that involve determining the cluster resources and MapReduce job-level parameter configurations to meet desired requirements on execution time and cost for a given analytic workload.

The Elastisizer is a novel system to which users can express their cluster sizing problems as queries in a declarative fashion. As we demonstrated in Section 8, the Elastisizer can provide reliable answers to these queries using an automated technique; providing nonexpert users with a good combination of cluster resource and job configuration settings to meet their needs. The automated technique is based on a careful mix of job profiling, estimation using black-box and white-box models, and simulation.

Multi-tenancy is a key characteristic of IaaS cloud platforms that can cause variations in the performance of MapReduce workloads. In other words, the execution time of a particular MapReduce job $j$ can vary based on what other MapReduce jobs and background processes run concurrently with $j$ in the cluster. We plan to study the effects of multi-tenancy and to enable the Elastisizer to recommend robust configuration settings for running a MapReduce workload under different conditions. This work will require extensions to the simulation process used by the What-if Engine. Another interesting avenue for future work is to add support for auction-based resource provisioning, e.g., spot instances on Amazon EC2.

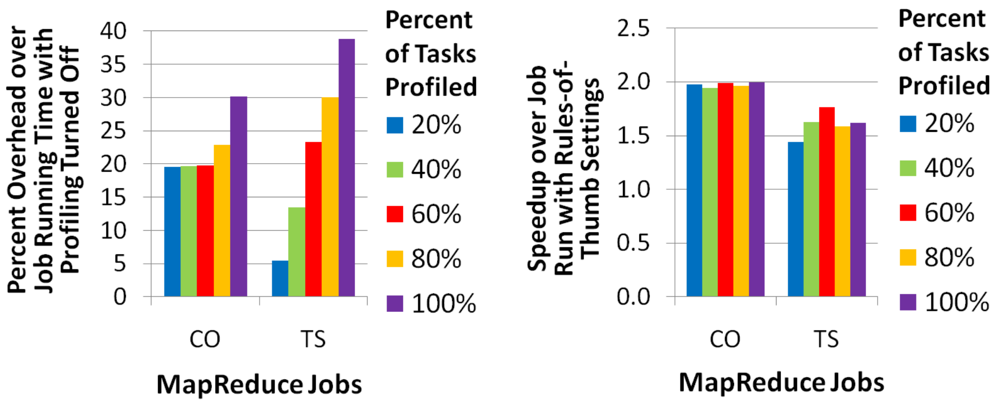


**Figure 13: (a) Overhead to measure the (approximate) profile, and (b) corresponding speedup given by the Elastisizer over the rules-of-thumb settings as the percentage of profiled tasks is varied for Word Co-occurrence (CO) and TeraSort (TS).**

## 10. ACKNOWLEDGEMENTS

We thank the anonymous reviewers for comments that helped us improve the paper significantly. We are extremely grateful to Amazon Web Services for awarding us a research grant for the use of resources on the Amazon Elastic Compute Cloud (EC2) and Elastic MapReduce.

## 11. REFERENCES

[1] *Amazon Elastic MapReduce*. http://aws.amazon.com/elasticmapreduce.
[2] S. Babu. Towards Automatic Optimization of MapReduce Programs. In *SOCC*, pages 137–142, 2010.
[3] P. Bodik, R. Griffith, C. Sutton, A. Fox, M. Jordan, and D. Patterson. Statistical Machine Learning Makes Automatic Control Practical for Internet Datacenters. In *HotCloud*, 2009.
[4] *Facebook on Hadoop, Hive, HBase, and A/B Testing*. http://tinyurl.com/3dsdsh4.
[5] *BTrace: A Dynamic Instrumentation Tool for Java*. http://kenai.com/projects/btrace.
[6] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic Instrumentation of Production Systems. In *USENIX ATEC*, 2004.
[7] S. Chaudhuri, P. Ganesan, and V. R. Narasayya. Primitives for Workload Summarization and Implications for SQL. In *VLDB*, pages 730–741, 2003.
[8] N. Chohan, C. Castillo, M. Spreitzer, M. Steinder, A. Tantawi, and C. Krintz. See Spot Run: Using Spot Instances for MapReduce Workflows. In *HotCloud*, 2010.
[9] *Cloudera: 7 tips for Improving MapReduce Performance*. http://www.cloudera.com/blog/2009/12/7-tips-for-improving-mapreduce-performance/.
[10] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51(1):107–113, 2008.
[11] S. Duan, V. Thummala, and S. Babu. Tuning Database Configuration Parameters with iTuned. *PVLDB*, 2(1):1246–1257, 2009.
[12] J. Hamilton. *Resource Consumption Shaping*. http://tinyurl.com/4m9vch.
[13] H. Herodotou. Hadoop Performance Models. Technical report, Duke Univ., 2010. http://www.cs.duke.edu/starfish/files/hadoop-models.pdf.
[14] H. Herodotou and S. Babu. Profiling, What-if Analysis, and Cost-based Optimization of MapReduce Programs. *PVLDB*, 4, 2011.
[15] H. Herodotou, H. Lim, G. Luo, N. Borisov, L. Dong, F. B. Cetin, and S. Babu. Starfish: A Self-tuning System for Big Data Analytics. In *CIDR*, pages 261–272, 2011.
[16] M.-Y. Iu and W. Zwaenepoel. HadoopToSQL: A MapReduce Query Optimizer. In *EuroSys*, pages 251–264, 2010.
[17] E. Jahani, M. J. Cafarella, and C. Ré. Automatic Optimization of MapReduce Programs. *PVLDB*, 4:386–396, 2011.
[18] K. Kambatla, A. Pathak, and H. Pucha. Towards Optimizing Hadoop Provisioning in the Cloud. In *HotCloud*, 2009.
[19] A. Li, X. Yang, S. Kandula, and M. Zhang. CloudCmp: Shopping for a Cloud Made Easy. In *HotCloud*, 2010.
[20] J. Lin and C. Dyer. *Data-Intensive Text Processing with MapReduce*. Morgan and Claypool, 2010.
[21] M. Mesnier, M. Wachs, R. Sambasivan, A. Zheng, and G. Ganger. Modeling the Relative Fitness of Storage. *SIGMETRICS*, 35(1):37–48, 2007.
[22] *Mumak: Map-Reduce Simulator*. https://issues.apache.org/jira/browse/MAPREDUCE-728.
[23] *OpenCore Probes vs Sun BTrace*. http://opencore.jinspired.com/?page_id=588.
[24] R. J. Quinlan. Learning with continuous classes. In *5th Australian Joint Conference on Artificial Intelligence*, pages 343–348, 1992.
[25] A. Qureshi, R. Weber, H. Balakrishnan, J. V. Guttag, and B. Maggs. Cutting the Electric Bill for Internet-scale Systems. In *SIGCOMM*, pages 123–134, 2009.
[26] G. Wang, A. R. Butt, P. Pandey, and K. Gupta. A Simulation Approach to Evaluating Design Decisions in MapReduce Setups. In *MASCOTS*, pages 1–11, 2009.
[27] T. White. *Hadoop: The Definitive Guide*. Yahoo Press, 2010.
[28] T. Ye and S. Kalyanaraman. A Recursive Random Search Algorithm for Large-scale Network Parameter Configuration. In *SIGMETRICS*, pages 196–205, 2003.
[29] W. Zheng, R. Bianchini, J. Janakiraman, J. R. Santos, and Y. Turner. JustRunIt: Experiment-Based Management of Virtualized Data Centers. In *USENIX ATC*, 2009.